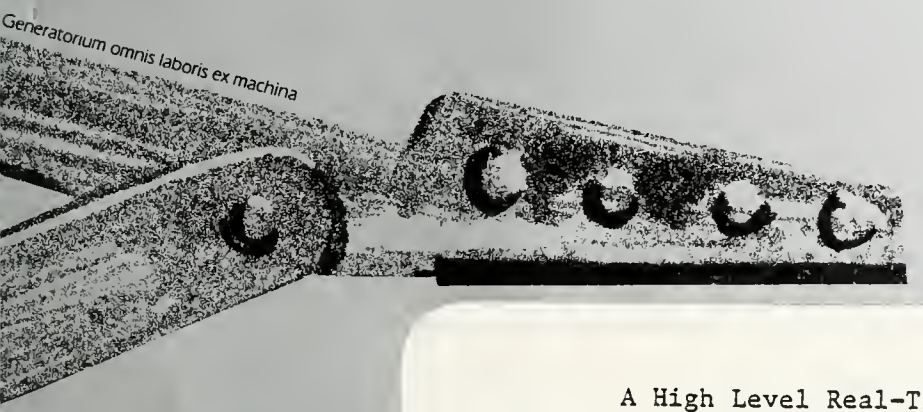
Robotics Research Technical Report



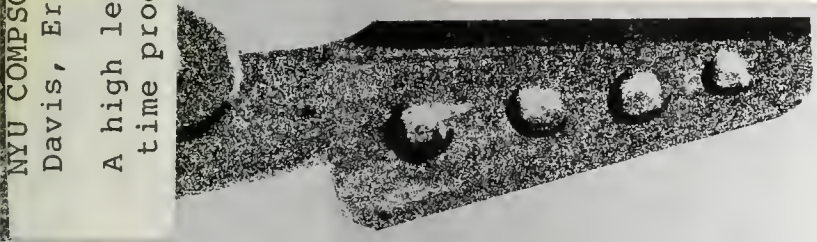
A High Level Real-Time
Programming Language

by

Ernest Davis

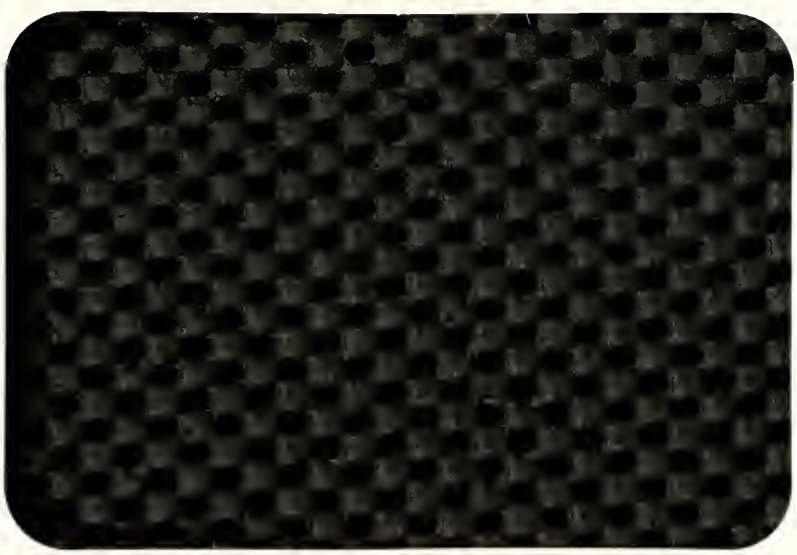
Technical Report No. 145
Robotics Report No. 36

October, 1984



New York University
Courant Institute of Mathematical Sciences

Computer Science Division
251 Mercer Street New York, N.Y. 10012



A High Level Real-Time Programming Language

by

Ernest Davis

Technical Report No. 145
Robotics Report No. 36

October, 1984

Work on this paper has been supported in part by a grant from the National Science Foundation under Proposal No. MCS-8402309.

A High Level Real-Time Programming Language

ABSTRACT

COAL is a high-level language for writing real-time programs, particularly aimed at robotics applications. Our presentation includes an informal description of the language, a formal semantics for most of the constructs, and a few examples of typical use. The most prominent features of COAL are variables whose value changes continuously with time, and extensive use of real-valued functions.

October 14, 1984

A High Level Real-Time Programming Language

1. Introduction

We present here an outline of a robot programming language, COAL (COntinuous Action Language), together with a formal semantics. COAL is designed to support an abstraction of programming which can operate in a continuous fashion. Variables can change values continuously over time, execution can be instantaneously interrupted in response to an event, and so on. Such constructs must be implemented in terms of discrete steps, but COAL expresses the idealization behind the implementation, in the way that real arithmetic is the idealization behind floating point arithmetic. In this paper, we will describe the basic elements of COAL, define a formal semantics, and present a few typical programming examples. We have not yet addressed the question of implementation.

2. Overview of COAL

2.1. Variables:

COAL supports at least the following types of variables: booleans, integers, real valued vectors (elements of R^n), continuous functions from R^m to R^n (called "real functions"), and integer semaphores.

Certain global variables will be bound to the input and output devices of the particular robotic system. *Sensor* variables are associated with particular sensors. At each moment in time, the value of the variable is the current measurement of the sensor. *Effector* variables are associated with parameters of the robot's effectors. Changing the value of the variable causes a motion by the robot. Effector variables must therefore be generally changed continuously. The global variable *clock* always has the current time as its value.

Anonymous variables include individual coordinates and time derivatives of vector variables. Thus, if V is a vector in R^3, then $V[1]$ is its x-component; V' is its time derivatives (when this derivative is finite); and $V[1]'$ is the derivative of the x-component.

2.2. Expressions

An expression may be a constant, a variable, or a function applied to expressions. Functions may be either built-in or user-defined. The built-in functions include at least the arithmetic, trigonometric, and exponential functions (with inverses); constructor functions and simple geometric operators for vector variables; differentiation and integration; and lambda abstraction and application.

2.3. Primitive Statements

There are two kinds of primitive statements: assignment statements, and synchronization primitives. For simplicity, we will use Dijkstra's "wait" and "signal" ("P" and "V") as the synchonization primitives. COAL differs from standard languages in that there are four different types of assignment: "assign", "bind", "follow", and "graph".

"Assign($V \leftarrow E$)" is the usual assignment statement. E is evaluated once and its value is stored in V. Thus "assign($X \leftarrow X+1$)" increments X by 1.

"Bind($V \leftarrow E$)" is continuous assignment, such as found in SERVOL (see [1]). The value of V is kept equal to the current value of E. Thus if P and T are sensor variables corresponding to pressure and temperature, then bind($X \leftarrow P*T$) keeps X equal to their product. Derivatives of a variable may be bound; if so, the bind statement must specify the initial conditions on the lower-order derivatives. For example "bind ($V' \leftarrow DT$; $V \leftarrow 5$)" fixes the derivative V' to the instantaneous value of DT, starting with $V=5$. "bind($V'' \leftarrow -k*V$; $V' \leftarrow 6$; $V \leftarrow 0$)" causes V to move in accordance with the harmonic equation, with an initial displacement of 0 and an initial velocity of 6. Some further restrictions on the use of "bind" are explained below.

"Follow($V \leftarrow E$)" is used to cause a variable to trace out, over time, a precomputed function. The value of E must be a particular time function f at the time the statement is invoked. The

effect of the statement is to cause the value of V at subsequent times T to be equal to $f(T)$. By convention, all the global variables in E are evaluated when the follow statement is entered. For instance, let X be an effector variable representing the robot's x-coordinate. If the statement "follow($X \leftarrow \lambda(T)(X+T-clock)$) is invoked at time T_0 with the robot at X_0, then the right hand side evaluates to the function $\lambda(T)(X_0+T-T_0)$. While the statement is active, the value of X is continually bound to the current value of this function of time. Since X is an effector variable, the effect is to move the robot forward at constant speed.

"Graph($V \leftarrow E$)" is used to save the behavior of a variable over time as a real function over time with no global variables. E is evaluated continuously. The value of V is a function which expresses the time variance of E from the time the statement is invoked until the time that it ends. For instance, let X be a sensor variable representing the x-coordinate of the robot, and suppose the robot, for whatever reason, moves forward at unit speed. If the statement "follow($X\_HISTORY \leftarrow X$)" is invoked at time T_0 when the robot is at location X_0 then *X_HISTORY* becomes the real function $\lambda(T)(X_0+T-T_0)$. The domain of X_HISTORY as a real function is the interval starting at T_0 and ending either at the current time, or at the time when the "follow" statement came to an end, whichever is first.

"Graph" can be used together with "follow" to implement robot guiding, whereby a robot learns a sequence of actions by being moved through them. The program would use a "graph" statement to record the desired motion in a variable V, and then a "follow" statement to reproduce them. "Graph" is somewhat analogous to the "history" statement in SERVOL [1].

In constrast to "assign", where the action can be thought of as being performed instantaneously, "bind", "follow", and "graph" require some finite interval of time for their performance. We define them as being active forever, except when this is modified by a "monitor" statement in the enveloping control structure, as will be explained below.

Circularities and multiple binding must be avoided in using "bind" and "graph". The statement "bind($X \leftarrow X+1$)" is meaningless, as is the concurrent execution of the statements "bind($Y \leftarrow X$), bind ($Y \leftarrow X+1$)" or the concurrent execution of the statement "bind($Y \leftarrow X$), bind ($X \leftarrow Y+1$)". It is

allowable to bind the derivative of a variable to be a function of a variable. For example "bind($X' \leftarrow X$)" causes X to increase exponentially with time. However it is not permitted to bind a variable to a function of one of its derivatives. There are two reasons for this restriction. Firstly, an equation of the form $X=f(X,X',X''...)$ is not guaranteed to have a solution. Secondly, proper differential equations, where the highest order derivative is express as a function of the lower orders, can be solved by simple interative methods: evaluate the function to find the value of the highest order derivative, and use that value to extrapolate the next values of the lower order derivatives. These methods do not converge when applied to equations of an improper form.

We may formulate these conditions as follows. Consider a graph whose nodes are all the variables and their derivatives. At any given moment of execution, construct an arc from variable A to B if A is a derivative of B, or if there is an bind or graph statement currently executing of the form "bind($A \leftarrow E(B)$)" or "graph($A \leftarrow E(B)$)", where $E(B)$ is an expression involving B. Then, at each instant, this graph must be acyclic. Moreover, if we merge the nodes of a variable with those of its derivatives then none of the resulting nodes has more than one arc leading to it. The correctness of these conditions must be checked at run-time each time a "bind" or "graph" statement is invoked. We do not have to include "follow" statement here, since the right hand of "follow" is not continuously reevaluated.

User-defined functions which are invoked on the right side of a "bind" or "follow" statement must be such that it makes sense to think of evaluating them instantaneously and continously. For example, if the function F had a side-effect of incrementing a global variable X by 1, then the statement "bind($V \leftarrow F()$)" would have to increment X infinitely often, which is not acceptable. We therefore define an *instantaneous* function as one where the only variables accessed are the arguments (passed by value) and local variables; and we require that only instantaneous functions be used on the right-hand side of a "bind" or "follow" statement.

"Follow" statements, like "bind" statements, may have as their left hand side either a variable or its derivative. "Assign" and "graph" must have a variable as a left hand side.

2.4. Control structures.

COAL provides five statement level control structures:

1) Sequencing. "sequence $[S_1; S_2; \ldots S_n]$" executes statements S_1, S_2, ... in order.

2) Conditionals. "if B then S_1 else S_2" evaluates the boolean expression B and executes S_1 if B is true, otherwise executes S_2.

3) Loops. "repeat S until B" iterates executions of the statement S until the boolean expression B becomes true at the end of an iteration.

4) Exception raising. "monitor B do S_1 else S_2" executes S_1 while continuously evaluating boolean expression B. If B ever becomes false, then the execution of S_1 is aborted, and S_2 is performed instead. A basic use of monitor statements is to bring an end to bind, follow, and graph statements.

5) Concurrency. "concurrent $[S_1; S_2; \cdots; S_n]$" executes statements S_1 through S_n concurrently.

This above list constitutes a minimal set of basic control primitives. It is not meant to exclude variants that clarify code through "syntactic sugar", such as loops with termination test at the end, one branch conditionals, case statements, etc. Indeed, we will introduce some additional such constructs in section 4.

2.5. Procedures and Functions.

Procedures and functions consist of a set of formal parameters, a set of local variables, and a body, which is a single statement. Functions have one local variable designated as the "return" variable whose value on exit from the body of the function is returned as the value of the function. Functions and procedures are strongly typed.

2.6. Pragmas

Since there is a considerable gap between the continuous idealization expressed by the language and its discrete implementation, it will be useful to define a number of optional methods of "giving advice" to the compilers, where standard defaults are not appropriate. The most

important such pragma would be an indication of how often a continuously evaluated expression should be reevaluated. Other pragmas would be the necessary accuracy of the real arithmetic or the appropriate representation for particular real functions.

3. Formal Semantics.

In this section, we describe a formal semantics for the primitive statements and control structures of COAL, ignoring the problems of expression evaluation, and procedure invocation. We specify the semantics of COAL constructs in terms of sets of *traces*. A trace is one possible behavior of the variable values, together with an explanation for that behavior. Formally, a trace is a quadruple $<I,H,F,N>$. I is a time interval of non-zero length that is closed on the left and either closed or infinite on the right. We will use T_l and T_u to designate the lower and upper bounds of I. H is a *history* of the variables; namely, a function from variables and time in I to the value of the variable at that time. We write $H(V,T)$ to mean the value of variable V at time T. For example, if variable FOO has value 5 at time $T=2.8$, then $H(\text{"FOO"},2.8)=5$. We extend H to expressions in the natural way; $H(\text{"FOO+4"},2.8)=9$.

F and N describe the sequence in which the primitive operations (the four assignment statements and the synchronization primitives) are executed. With each separate execution of each primitive operation we associate an individual integer. The value of the integer is irrelevant; all that matters is that different executions have different integers. N is the function which associates a given integer with a given statement being executed. F is the function which specifies, for each instant of time, which executions are taking place. For example, it might be the case that at time $T=2.8$, executions numbers 218 and 556 are taking place; that 218 is an execution of the statement "wait(QUEUE1)"; and that 556 is the statement "bind(Y←X+1)". Then we would have $F(2.8)=\{218, 556\}$, $N(218)=$"wait(QUEUE1)"; and $N(556)=$"bind(Y←X+1)". If these same statements are executed again at later times, these new executions would have different numbers. Note that $N(F(T))$ is always the set of statements being executed at time T. We say that F and N together constitute the *execution sequence* for the history H.

The semantics of an COAL program is expressed in terms of all the traces possible in executing that program. We build up the semantic definition of COAL inductively, first defining the traces for the primitive operations, and then combining these to create the traces for the control structures. There is a difficulty, however. We would like to define the semantics of an assignment statement, for example, so that no variables change their value except the one being assigned. However, if there are concurrent operations, this may not be the case; other variables may change their values as a result of other processes. We therefore define two kinds of traces. A *performance* of the program is a trace in which everything that should happen does happen, but other things may happen as well. This corresponds to running the program with arbitrary other concurrent processes running as well. An *execution* of a program is a trace where only those things that must happen do happen, corresponding to running the program with no other concurrent processes. We define the semantics by building the structure of performances inductively, and then, at the end, imposing the execution condition.

To specify the connection between the sensor and effector variables and the real world would require a larger semantic theory that includes real world statements. In terms of the internal programming language, the values of sensor variables are entirely arbitrary.

We will assume of all variables that their values, as functions of time, are everywhere continuous from the left, and, in fact, infinitely differentiable from the left. Moreover, the set of points where any variable or derivative has a discontinuity from the right does not have any cluster point; that is, there are only finitely many such points in any finite interval. It is, of course, possible to write COAL programs which violate these conditions; for example, to write infinite loops where each iteration takes half as long as the last. Such programs are considered semantically invalid.

In our semantic definition, we assume a non-standard model of the real numbers, with infinitesimals. In real time programming, it is generally useful to assume, at least as a first-order approximation, that all calculations can be performed before anything has time to happen in the real world. On the other hand, we do not want to assume that such calculations take identically

zero time, because we want to specify the order of execution in terms of increasing starting times. The non-standard model of the reals provides an elegant solution: calculations take infinitesimal time. We use a fixed infinitesimal quantity δ in defining the semantics of both the assignment and monitor statement.

In all that follows, V^n is the nth derivative of V. In particular, $V^0 = V$.

We now proceed with our formal definitions:

Definition 1:

A performance of the statement S="assign($V \leftarrow E$)" is the set of traces $<I=[T_l,T_u],H,F,N>$, such that there exist a time $T_a \epsilon I$ satisfying the following:

$T_u - T_l$ is infinitesimal;

$T_a + \delta \leq T_u$;

for all $T \epsilon (T_a, T_a+\delta]$, $H(V,T)=H(E,T_b)$;

for all $T \epsilon I$, $N(F(T))=\{S\}$.

In other words, at some point T_a in I, E is evaluated, and its value is assigned to V. V retains this value for at least time δ. The whole assignment takes an infinitesimal time; then the program may move on to the next statement. Finally, the execution sequence F,N asserts that the assignment statement is active during the whole interval. The reason we do not simply consider I to be the interval $[T_a,T_a+\delta]$ is to allow for the possibility of concurrent assignments; the actual assigning of one statement may occur during the unspecified part of the interval of the other. We require that V retain its new value for at least time δ in order to allow the definition of "monitor" to work with it (see below). Note that the values of all other variables are completely unconstrained, to allow for the possibility that concurrent processes are affecting them.

Definition 2:

A performance of the statement S="bind($V^n \leftarrow E_n; V^{n-1} \leftarrow E_{n-1}; \cdots V \leftarrow E_0$)" is the set of traces $<I=[T_l,\infty),H,F,N>$, such that there exists a time $T_a \epsilon I$ such that

$T_a - T_l$ is infinitesimal;

for all $(T>T_a)$, $H(V^n,T)=H(E_n,T)$;

for all $(m<n)$, $H(V^m,T_a)=H(E_m,T_l)$,

for all $(T \epsilon I)$, $N(F(T))=\{S\}$.

Inherantly, a bind statement is active forever; this may be modified by a containing monitor statement. We specify that all during this interval the highest derivative is bound to the corresponding expression, and that the lower derivatives are set to their starting values infinitesimally soon after the beginning of the interval.

Definition 3:

A performance of the statement S="follow($V^n \leftarrow E_n; V^{n-1} \leftarrow E_{n-1}; \cdots V \leftarrow E_0$)" is the set of traces $<I=[T_l,\infty),H,F,N>$, such that there exists a time $T_a \epsilon I$ such that

$T_a - T_l$ is infinitesimal;

$H(E_n,T_l)$ evaluates to the real function $e(T)$; and for all $(T>T_a)$, $H(V^n,T)=e(T)$;

for all $(m<n)$, $H(V^m,T_a)=H(E_m,T_l)$,

for all $(T \epsilon I)$, $N(F(T))=\{S\}$.

Here we specify that E_n is evaluated as a real function at the beginning of I, and that V^n follows the course specified by that value.

Definition 4:

A performance of the statement S="graph($V \leftarrow E$)" is the set of traces $<I=[T_l,\infty),H,F,N>$, such that there exists a time $T_a \epsilon I$ such that

$T_a - T_l$ is infinitesimal;

for all $(T>T_a)$ $H(V,T)$ evaluates to a time function $v(T)$ such that,

for all $T' \epsilon (T_l,T]$, $v(T')=H(E,T')$; and

for all $(T \epsilon I)$, $N(F(T))=\{S\}$.

Here we require that V continuously evaluate to a function which expresses the whole history of E from T_l until now. We can abbreviate the main condition above, if $T>T'>T_a$, then $(H(V,T))(T')=H(E,T')$.

Finally, we have the semantics for the two semaphores:

Definition 5:

A performance of the statement $S=$"wait(V)" is the set of traces $<I=[T_l,T_u],H,F,N>$, where one of two cases applies:

1) (Normal Case) There exists $T_b \epsilon I$ such that

$T_u - T_b$ is infinitesimal;

$H(V,T_b)>0$;

$H(V,T_u)=H(V,T_b)-1$;

there is no subinterval of I of finite length in which $H(V,T)=0$; and

for all $T \epsilon I$, $N(F(T))=\{S\}$.

2) (Eternal Suspension)

I is infinite;

there is no subinterval of I of finite length in which $H(V,T)=0$; and

for all $T \epsilon I$, $N(F(T))=\{S\}$.

Thus the process waits until V is positive, and then decrements it by 1 and proceeds.* In the meantime, other processes can be manipulating V with waits and signals; but V should never remain positive for a finite (non-infintesimal) length of time without our process taking advantage of the fact. The restrictions against two processes accessing the same semaphore simultaneously will be built into our definition of "execution".

* There are several possible definitions for "wait" and "signal" given in the literature. The one used here, from [4], was chosen to simplify this definition.

Definition 6:

A performance of the statement S="signal(V)" is the set of traces $<I=[T_l,T_u],H,F,N>$, where

$T_u - T_l = \delta$;

$H(V,T_u)=H(V,T_l)+1$; and

for all $T \epsilon I$, $N(F(T))=\{S\}$.

This completes the semantics for the primitive operations. The semantics for the control statements are derived by combining the semantics for their arguments.

Definition 7:

Two functions N_1 and N_2 are said to be *joinable* if their domains are disjoint. N is the *join* of N_1 and N_2 if the domain of N is the union of the domains of N_1 and N_2, and N agrees with each on its own domain.

Definition 8:

Trace $P_2=<I_2=[T_{2,l},T_{2,u}],H_2,F_2,N_2>$ *fits after* trace $P_1=<I_1=[T_{1,l},T_{1,u}],H_1,F_1,N_1>$ iff N_1 and N_2 are joinable; $T_{1,u}=T_{2,l}$; and, for all variables V, $H_1(V,T_{1,u})=H_2(V,T_{2,l})$. Thus, the end of the first is the same as the beginning of the second.

Definition 9:

Given a series of traces $P_1=<I_1=[T_{1,l},T_{1,u}],H_1,F_1,N_1>$, $P_2=<I_2=[T_{2,l},T_{2,u}],H_2,F_2,N_2>$, $\cdots$ $P_n=<I_n=[T_{n,l},T_{n,u}],H_n,F_n,N_n>$, such that P_{k+1} fits after P_k, we define their sequence as the trace $P=<I=[T_l,T_u],H,F,N>$ = sequence($P_1,P_2,\ldots,P_n$) such that

$T_l=T_{1,l}$ and $T_u=T_{n,u}$;

for all ($T \epsilon [T_{k,l},T_{k,u}]$) for all variables V, $H(V,T)=H_k(V,T)$;

N is the join of $N_1 \cdots N_n$;

$F(T_{k,u})=F_k(T_{k,u}) \bigcup F_{k+1}(T_{k+1,l})$; and

for all $(T \epsilon (T_{k,l},T_{k,u}))$, $F(T)=F_k(T)$;

Thus, the sequence is formed by stringing together the separate traces.

Defining the semantics of the control structures is now straightforward. To shorten the definitions, we will assume the notational convention that, for any subscript q, the performance $P_q=<I_q=[T_{q,l},T_{q,u}],H_q,F_q,N_q>$.

Definition 10:

Let statement S = "sequence $[S_1; S_2; \ldots S_n]$". A performance of S is a trace P such that there exists performances $P_1,P_2, \ldots ,P_n$ of $S_1,S_2, \ldots ,S_n$ respectively such that P_{k+1} fits after P_k and P=sequence$(P_1,P_2, \ldots ,P_n)$

Definition 11:

Let statement S = "if B then S_1 else S_2". The trace $P=<[T_l,T_u],H,F,N>$ is a performance of S if either $H(B,T_l)=TRUE$ and P is a performance of S_1 or $H(B,T_l)=FALSE$ and P is a performance of S_2.

Definition 12:

Let statement S = "repeat S_1 until B". The trace P is a performance of S if either there exists a finite sequence of performances $P_1,P_2, \ldots ,P_n$ or an infinite sequence of performances $P_1,P_2, \cdots$ such that

P=sequence$(P_1,P_2, \cdots)$;

each P_k is a performance of S_1;

P_{k+1} fits after P_k;

for $k<n$, $H_k(B,T_{k,u})=FALSE$;

if the sequence is finite, then $H_n(B,T_{n,\mu})=TRUE$; and

if the sequence is infinite then the series $T_{1,\mu},T_{2,\mu},\ldots$ diverges.

The final condition, that the ending times diverge if the loop is infinite, prevents us from performing infinitely many loop iterations in finite time. To do so would create a number of problems. Any COAL program which forces infinitely many iterations in finite time is considered semantically improper.

Definition 13:

Let statement S = "monitor B do S_1 else S_2". The trace P is a performance of S if one of the following two conditions is true:

1) P is a performance of S_1, and for all $T\epsilon I$, $H(B,T)=TRUE$;

2) there exists traces P_1, P_2 and P_a such that

P_1 is a performance of S_1 and P_2 is a performace of S_2;

$T_{a,l}=T_{1,l}$; $T_{a,\mu}\leq T_{1,\mu}$;

for all $(T\epsilon I_a)$ $N(F_a(T))=N(F_1(T))$

for all variables V, $H_a(V,T)=H_1(V,T)$;

$P=\text{sequence}(P_a,P_2)$;

for all $T<T_{a,\mu}-\delta$, $H_a(B,T)=TRUE$; and

for some $T\epsilon(T_{a,\mu}-\delta,T_{a,\mu})$, $H_1(B,T)=FALSE$.

In the first case, B is always true, and so S_1 executes without interruption. In the second case, B becomes false at some time infinitesimally close to time T. In this case, P is divided into two parts. The first part, P_a, is a partial execution of P_1, a performance of S_1. Thus, P_a agrees in all respects with P_1 up until T_a. The second part, P_2, is a performance of S_2.

The use of the same δ here as in the assignment statement is to guarantee that if a monitor is triggered by the effect of an assignment statement, then the monitor will halt

execution before the program passes onto the next statement.

In the case where one monitor is imbedded in another with the same halting condition, the definition above is non-deterministic as to whether the exception handler of the inner monitor begins execution. It certainly does not execute for more than time δ.

Definition 14:

Let S be the statement "concurrent $[S_1; S_2; \cdots S_n]$". $P=<I,H,F,N>$ is a performance of S if there exist $F_1,F_2 \cdots F_n$ such that $<I,H,F_k,N>$ is a performance of S_k and, for all $T \in I$, $F(T)=F_1(T) \bigcup F_2(T) \bigcup \cdots \bigcup F_n(T)$.

Thus, the history of a concurrent action must be a history of each of its components; everything that must happen in each component separately must happen when they are run together. The execution sequence for the concurrent action is the union of the separate actions.

Sometimes there will be no history at all which is a history of all the components. This happens, for example, when conflicting "bind" statements are made concurrent. There is no history which is part of performances of both "bind($X \leftarrow Y$)" and "bind($X \leftarrow Y+1$)". In this case, the statement is considered semantically invalid. No behavior whatever is consistent with this statement. A subtle problem with this approach is discussed in section 3.1.

This completes our definition of the performance of a statement. We are now ready to define the *execution* of a statement. Conceptually, an execution is a performance in which every change of every variable can be explained in terms of the execution sequence, and in which the synchronization constraints are observed.

Definition 15:

Let S be a statement and let $P=<I,H,F,N>$ be a performance of S. P is an execution of S iff the following conditions are observed:

1) Let $I'=[T'_0,T'_1]$ be any subinterval of I, and let V be any variable or derivative. If $H(V,T'_0) \neq H(V,T'_1)$ then either V is a sensor variable or one of its derivatives, or there

exists a time $T \epsilon I'$ and a statement $S \epsilon N(F(T))$ [redacted] where S is an "assign", "bind", "follow", "graph", "wait", or "signal" statement with V or a derivative of V on the left hand side. Thus, we ensure that no variable changes without a good reason

2) Let e_1 and e_2 be two integers representing executions of primitive operations in the domain of N such that $N(e_1)$ and $N(e_2)$ are both statements of the form "wait(V)", with the same semaphore s. Let $I_1=[T_{1,l},T_{1,u}]=F^{-1}(e_1)$, and $I_2=[T_{2,l},T_{2,u}]=F^{-1}(e_2)$. Thus I_1 and I_2 are the intervals when these particular executions take place. Then $T_{1,u}$ and $T_{2,u}$ differ by at least δ. Given the semantics of performances of "wait", this ensure that no two processes continue past the wait statement simultaneously.

3) Let e_1 and e_2 be two executions of statements of the form "signal(V)". Let $I_1=F^{-1}(e_1)$ and $I_2=F^{-1}(e_2)$ be the intervals of execution, as above. Then I_1 and I_2 are disjoint. Thus, no two processes can execute "signal" statements simultaneously.

3.1. A Problem with Concurrency.

The above definition of concurrent actions has a serious, though subtle, bug: it forces computers to predict the future. The problem is that, if there are two ways of performing the beginning of some concurrent actions, and one of these leads to a dead end in the form of inconsistent bindings, then the semantics simply throws out this path and requires the computer to take the other path. However, there is no way for the computer to know which path, if either, will get it into these straits; hence, the semantics cannot be implemented.

An example will clarify the problem. Consider the following program:

```
sequence
  [ assign (B ← TRUE);
    concurrent
      [ monitor (B) bind (X ← 1) else bind (Y ← 1);
        sequence [ wait (S); assign (B ← FALSE); signal (S); bind (Z ← 1) ];
        sequence [ wait (S); bind (X ← 2); signal (S)]
      ]
  ]
```

Thus, the boolean flag B is set true and then three concurrent operations are set in motion. The first binds X to be 1 while B is true, and then does something else. The second sets B to be false in a section protected by the semaphore S. The third binds X to 2 in a section protected by the semaphore S.

In running this code on an actual computer, one would expect one of two behaviors. If the "wait" from the second concurrent action is executed first, then B will be set to FALSE, the binding of X to 1 will cease, the signal will be executed, the third concurrent action will be allowed to proceed, and X will be bound to 2. If, however, the third action is given priority, then the program immediately enters into an inconsistent state where X is bound to both 1 and 2. But in the semantics which I have given, this last is not a behavior of any kind. The only valid behavior is the first. Thus the program must follow the first sequence, which is not a reasonable expectation.

Intuitively, one would like to say that, if there is more than one performance for the initial part of a program, then all of these are legitimate, and if it gets into trouble later, it enters a "bombout" state. The difficulty is in defining an initial part of a performance which, as a whole, does not exist. A similar problem with regard to physical actions was solved in [2] by considering motions which were physically impossible and to speak of actually carrying out as much of them as are physically possible. The analogous solution here would be to define trace which are inconsistent and to say that, if an intial part of such a trace is consistent, then it represents a performance of part of the program. One possible approach

might be to define "illegal" traces which give multiple values to variables. We have not developed this approach in detail.

4. Examples

A few examples will help to illustrate the use of these COAL constructs. To begin, we will define some additional control structures which are, strictly speaking, redundant but useful in practice.

(1) A useful minor variant is the one-branch monitor "monitor *B* do *S*". This may be defined as "monitor *B* do *S* else no-op" where no-op can be defined as an assignment to a dummy variable.

(2) The construct "delay until *B*", for a boolean expression *B*, does just what it suggests; it monitors *B* until it becomes true, and then continues. Using a dummy variable *DUMMY* we may define wait as "monitor (not *B*) do bind(*DUMMY*←1)"

A similar construct is "delay for *T*", which causes the program to suspend itself for time *T*. This may be defined as

"sequence [assign (starttime ← clock);

monitor (clock < starttime + T) do bind (*DUMMY*←1)]"

(3) It is often useful to have a variable which keeps track of the time elapsed since a given starting time. We express this using the statement "stopwatch (*W*, *U*)", use *W* as a stopwatch variable, using *U* as a unit of time. We can define this in terms of our primitives as "bind(*W*'←*U*;*W*←0)" (though in a real system, this construct would almost certainly have some particular efficient implementation).

(3) A timed loop "every *T* do *S* until *B*" iterates executions of *S* every *T* units of time until *B* becomes true. For example, the statement "every (0.1 second) do fire until dead" causes the firing of 10 shots a second until it is detected that the target is dead. We may define this loop using the "repeat" loop as follows:

"repeat sequence [assign (W ← clock); S; delay until (clock > W + T)]

until *B*"

(5) In world modelling, it is often useful to say that one object moves with another. One way to state this is that their local frames of reference transform in the same way. AL [5] provides the "AFFIX" primitive to effect this. This can be defined in COAL as follows:

```
sequence [ assign (rel_pos ← find_transform(frame1, frame2));
           bind (frame2, transform(rel_pos, frame1))
         ]
```

"Frame1" and "frame2" are the two object frames. "Transform" applies a transform to a frame of reference. "Find_transform" finds the transform between two given frames of reference.

For our first full example,* consider raising a robot arm to touch the ceiling. Assume that the robot has feedback giving the current altitude of the arm and that he has direct control over the speed of the motor raising the arm. Also, assume that the speed of the motor must be changed continuously. In order to avoid bumping the arm, we would like it to slow down as it approaches the ceiling. We therefore establish an intermediate height "control". Up to "control", the arm will move at maximum speed. Past this height, it will gradually slow down until it approaches the ceiling at zero speed. We can program this as follows:

Example 1:

```
var altitude : real sensor;          /* Global variables */
    motorspeed : real effector;

procedure raise-arm (ceiling, control, maxspeed : real);
const maxspeed = 100; /* Units are fixed by the system */
var decelerate : real;
sequence
  [ monitor (altitude < control) do bind (motorspeed ← maxspeed);
    assign (decelerate ← maxspeed * 2 / (2 × (ceiling - control));
    monitor (motorspeed > 0) bind (d/dt(motorspeed) ← decelerate)
  ]
```

If the altitude is a directly controlable effector variable, we can achieve the same effect by substituting "d/dt(altitude)" for "motorspeed" throughout the above code.

* This is modified from an example in [1].

As another example, we will write a program to move a robot along a given path. The path is expressed as a parametrized function of arclength in two dimensions. We will assume that the robot should move at constant speed; that he should always point forward along the path; and that his orientation and position are directly controlable effector variables.

Example 2

```
var position : effector real [2];
    orientation : effector real;

procedure followpath (path : realfun (real → real × real));
const robotspeed = 100;
var  arclength : real;
concurrent
   [ stopwatch (arclength, robotspeed);
     bind (position ← apply (path, arclength));
     bind (orientation ← atan2 (d/dt(position [2]), d/dt(position [1]))
   ]
```

"Apply" in the code above is functional application: "apply(f,x)" is f(x).

If, instead, the speed and angular velocity were the directly controlable parameters, then we could modify the above code as follows:

Example 3

```
var speed : effector real;
    angspeed : effector real;

procedure followpath2 (path : realfun (real → real × real));
const desiredspeed = 100;
var arclength : real;
concurrent
  [ stopwatch (arclength, desiredspeed);
    bind (speed ← desiredspeed);
    bind (angspeed ← curvature (path, arclength))
  ]
/* Curvature (p, s) is a instantaneous function which computes
the curvature of a curve p parametrized by arclength at a point s. */
function curvature (p : realfun (real → real [2]); s : real) instantaneous;
var curve : return real; /* return variable */
    dp : real [2]; /* tangent to p at s */
    d2p : real [2]; /* normal to p at s */
sequence
  [ assign(dp ← derivative (p, s, 1); /* 1st derivative of p at s */
    assign(d2p ← derivative (p, s, 2); /* 2nd derivative of p at s */
    assign(curve ← dp[1] × d2p[2] - dp[2] × d2p[1])
  ]
```

If we wished to add to the first version of "followpath" the restriction that the robot could only move when the boolean flag "go" was set, we could rewrite it as follows:

Example 4

```
var position : effector real × real;
    orientation : effector real;
    go : sensor boolean;

procedure followpath3 (path : realfun (real → real × real));
const robotspeed = 100;
var arclength : real;
sequence
 [ assign (arclength ← 0.0);
   repeat
     monitor (go)
       do concurrent
            [ bind (d/dt(arclength) ← robotspeed);
              bind (position ← apply (path, arclength));
              bind (orientation ←  atan2 (d/dt(position [2]), d/dt(position [1])))
            ]
       else delay until go;
 ]
```

If the robot were following another robot, whose position and orientation it could sense, we could use the "graph" and "follow" functions as follows:

Example 5

```
var pos2 : sensor real [2]; /* position of the other robot */
    orient2 : sensor real;
    position : effector real [2];
    orientation : effector real;

procedure followrobot;
const dtime = 60;  /* delay time between our robot and the other */
var posfun2 : realfun (real → real [2]);
    angfun2 : realfun (real → real);
concurrent
  [ graph (posfun2, pos2);
    graph (angfun2, orient2);
    sequence
       [ delay for dtime;
         concurrent
           [ follow (position, lambda (T) (apply (posfun2, (T - dtime))));
             follow (orientation, lambda (T) (apply (angfun2, (T - dtime))));
           ]
       ]
  ]
```

5. Related work

Previous general purpose real-time programming languages, such as PEARL [6], MODULA [10], or GAELIC [3] have provided such facitilies as programming and synchronizing parallel tasks, timing initiation of processes to a real-world clock, halting processes on

interrupts, and non-standard I/O. Robot programming languages tend to provide these same control structures, together with more or less general geometric primitives for describing positions and motions of robot effectors or external objects. (See [5] for an overview.) For example, AL [7] provides these capacities in the context of a PASCAL-like language. AL also provides an AFFIX function, which has some of the capacities of COAL's "bind". MINI [8] extends LISP with a few basic robot functions. These can be combined with ordinary LISP code to provide a wide range of robot programmability. PAL [9] allows the user to specify successive positions of the end-effector in terms of transformational equations, and itself plans the motion between these positions.

More similar to COAL is SERVOL [1]. SERVOL provides both continuous and sequential assignment, equivalent to COAL's "bind" and "assign"; the capacity of keeping the recent past history of a variable, which gives some of the capacities of COAL's "graph". Exception raising can be simulated to a degree using conditional expressions and continuous assignment. SERVOL is designed to run the inner levels on a microcomputer communicating with a mainframe host. The syntax and semantics of the language are therefore simple and not very general.

The primary innovations of COAL are its use of continuous-valued variables in a uniform way, and its use of functions of real arguments as basic data types. The ease with which a formal semantics may be defined for COAL is indicative of the cleanness of the underlying theory.

6. Acknowledgements

Thanks to Paul Spirakis for his extensive comments and to Leora Morgenstern, Colm O'Dunlaing, and Ed Schonberg for their help.

7. Bibliography

[1] Bernstein, H.J, P.G. Lowney, and J.T. Schwartz, "SERVOL: Preliminary Proposal for a Programming Language for Real-Time Servo Control" Technical Report #120, Courant Institute.

[2] Davis, Ernest, "An Ontology for Physical Action", Technical Report #123, Courant Institute

[3] le Calvez, F., F. Madaule, and H.G. Mendelbaum "Compiling GAELIC: A Global Real Time Language" *Annual Review in Automatic Programming* Vol. 8, 1977 pp. 37-45

[4] Ledgard, H. and M. Marcotty *The Programming Language Landscape* SRA, Chicago, 1981

[5] Lozano-Perez, T. "Robot Programming" AI Memo #698, MIT AI Lab, December 1982

[6] Martin, T. "Realtime Programming Language PEARL - Concept and Characteristics", IEEE COMPSAC 1978, pp. 301-305

[7] Mujtaba, S. and R. Goldman "AL user's manual" AI memo #323, Stanford AI Lab, January 1979

[8] Silver, D. "The Little Robot System" AI Memo #273, MIT AI Lab, January 1973

[9] Takase, K., R.P. Paul, and E.J. Berg "A Structured Approach to Robot Programming and Teaching" IEEE COMPSAC, November 1979

[10] Wirth, N. "Modula: A Language for Modular Multi-Programming" *Software - Practice and Experience 7* 1977 pp. 3-35

NYU COMPSCI TR-145c.2
Davis, Ernest
A high level real-
time programming

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012